Eliminating Useless Messages in Write-Update Protocols on Scalable Multiprocessors

R. Bianchini, T.J. LeBlanc, and J.E. Veenstra

Technical Report 539
November 1994

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Eliminating Useless Messages in Write-Update Protocols on Scalable Multiprocessors

Ricardo Bianchini, Thomas J. LeBlanc, and Jack Veenstra

{ricardo,leblanc,veenstra}@cs.rochester.edu

Department of Computer Science
University of Rochester
Rochester, New York 14627
(716) 275-5426

Technical Report 539

October 1994

Cache coherence protocols for shared-memory multiprocessors use invalidations or updates to maintain coherence across processors. Although invalidation protocols usually produce higher miss rates, update protocols typically perform worse. Detailed simulations of these two classes of protocol show that the excessive network traffic caused by update protocols significantly degrades performance, even with infinite bandwidth. Motivated by this observation, we categorize the coherence traffic in update-based protocols and show that, for most applications, more than 90% of all updates generated by the protocol are unnecessary. We identify application characteristics that generate useless update traffic, and compare the isolated and combined effects of several software and hardware techniques for eliminating useless updates. These techniques include dynamic and static hybrid protocols, false sharing elimination strategies, and coalescing write buffers. Our simulations show that software caching (where coherence is managed under programmer or compiler control) and the dynamic hybrid protocol reduce useless updates the most, but coalescing write buffers produce fewer, albeit larger, coherence messages. As a result, coalescing write buffers usually produce the best running time, except when the block size is large or the bandwidth is limited. Finally, based on the observation that the techniques we consider are unable to eliminate a large number of useless updates, we suggest directions for further reducing the useless traffic in update-based protocols.

This research was supported under ONR Contract No. N00014-92-J-1801 (in conjunction with the ARPA HPCC program, ARPA Order No. 8930) and NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724. Ricardo Bianchini is supported by Brazilian CAPES and NUTES/UFRJ fellowships.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1994	technical report

4. TITLE AND SUBTITLE

Eliminating Useless Messages in Write-Update Protocols on Scalable Multiprocessors

5. FUNDING NUMBERS

ONR N00014-92-J-1802, ARPA 8930

6. AUTHOR(S)

R. Bianchini, T.J. LeBlanc, and J.E. Veenstra

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)

Office of Naval Research
Information Systems
Arlington VA 22217

ARPA
3701 N. Fairfax Drive
Arlington VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TR 539

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

invalidations; updates; scalable multiprocessors; coherence protocols

15. NUMBER OF PAGES

30 pages

16. PRICE CODE

free to sponsors; else $2.00

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

1 Introduction

The overhead of remote memory accesses is a major impediment to achieving good application performance on modern large-scale shared-memory multiprocessors. As processor speeds continue to improve at a dramatic rate, and as we anticipate building ever-larger machines, the relative importance of remote accesses will continue to grow. Shared-memory multiprocessors use hardware caches to reduce the average cost of a data access by storing data close to processors that need it. The cache coherence protocol determines how data is moved among the caches in the machine, ensuring that data is frequently found in the local cache, while preventing processors from using stale data.

There are two common classes of coherence protocol used in shared-memory machines: write-update protocols (WU) [McCreight, 1984; Thacker and Stewart, 1987; Thacker *et al.*, 1992] and write-invalidate protocols (WI) [Goodman, 1983; Papamarcos and Patel, 1984; Lenoski *et al.*, 1990]. Under a WU protocol, each time a processor writes shared data, the coherence protocol broadcasts the new value to every other processor caching that data. Under a WI protocol, a write to a shared cache block causes the coherence protocol to mark as invalid all other cached copies of the block.

The advantage of WU is that each processor receives and stores the update as it occurs, thus preventing future cache misses when the new value is needed. This property is particularly helpful when many processors read the updated values between successive write operations to the data [Eggers and Katz, 1988]. The disadvantage of WU is that every write operation to shared data requires that updates be sent over the network, even if no processor accesses the data between successive writes. WI achieves superior performance when cache blocks are written many times by a single processor before being accessed by any other processor [Eggers and Katz, 1988]. In most cases, WI results in higher miss rates, but fewer communication operations.

Previous studies comparing WI and WU protocols on bus-based machines have offered mixed results [Archibald and Baer, 1986; Eggers and Katz, 1988; Veenstra and Fowler, 1994]. In general, the comparison depends on the relative cost of reads and writes, and the sharing patterns exhibited by programs. WI protocols are currently used in the vast majority of hardware-coherent systems however. Although the broadcast nature of a shared bus allows the coherence protocol to update many processors with a single transaction, WI still performs better than WU in most cases because (1) the communication bandwidth available per processor on these machines is usually very limited and is rapidly consumed by the excessive number of transactions produced by WU and (2) the cost of an update transaction on the bus is roughly the same as the cost of rereading an invalidated cache block, and there are likely to be many more transactions under WU than under WI.

Scalable, network-based machines (such as the Stanford DASH [Lenoski *et al.*, 1992]) offer a very different environment for comparing WU and WI. These machines may incorporate relaxed consistency [Lenoski *et al.*, 1990] and write buffers (which reduce the cost of writes), or may use page-based coherence [Bisiani and Ravishankar, 1990; Carter *et al.*, 1991; Wilson and LaRowe, 1992] (which results in high latency and bandwidth requirements for page faults).

In this paper, we focus on scalable cache-coherent multiprocessors. Future machines in this class will likely have very high communication bandwidth and remote access latency. Under these architectural assumptions, one might expect WU to perform better than WI, because the higher miss rate of WI would likely have a greater negative impact on performance than the extra traffic associated with WU.

Our evaluation of WI and WU protocols as a function of bandwidth and block size does not confirm this expectation, however. Detailed simulations of these two classes of protocol show that the excessive network traffic caused by update protocols significantly degrades performance, even with infinite bandwidth. We trace the poor performance of WU to a variety of factors that are independent of bandwidth, but are all related to an excessive number updates. Motivated by this observation, we categorize the coherence traffic generated by WU protocols to quantify the amount of update traffic necessary for correct execution. The results of this analysis show that, for most applications, more than 90% of all updates are useless. Our analysis also pinpoints the application characteristics responsible for useless update traffic.

The dominance of useless traffic in our experiments led us to consider the extent to which techniques for improving the performance of WU protocols reduce useless traffic. We study several software and hardware techniques, such as dynamic and static hybrid invalidate/update protocols, false sharing elimination strategies, and coalescing write buffers. Although all the techniques we consider significantly reduce the useless traffic associated with update-based protocols, there is still the potential for further reductions in traffic. We suggest several directions for further eliminating useless traffic in update-based protocols.

Our work differs from most previous studies on coherence protocols in that our main goal is not to determine whether to use WI or WU for a particular architecture. Rather, in the same vein as [Dubois *et al.*, 1993], which examined useless misses in WI protocols, our main goals are to categorize updates in terms of their usefulness, and to compare techniques intended to improve the performance of WU protocols with respect to our categorization. In addition, we quantify the impact on execution time of these techniques, and the effect of bandwidth and block size on our results.

Our work is similar to recent work by Dahlgren *et al* [Dahlgren *et al.*, 1994; Dahlgren and Stenstrom, 1994] in that we consider some of the same techniques for improving WU protocols. In that work, write caches and dynamic hybrid protocols achieved better performance than WI. Our work is distinguished by our focus on the source and usefulness of updates (and their associated acknowledgements), which allows us to relate program modifications to protocol improvements, evaluate protocol optimizations with respect to reductions in network traffic, and suggest new optimizations to update-based protocols. Furthermore, by treating both bandwidth and block size as parameters, and by considering several additional modifications to a pure WU protocol, we explore interactions between these factors not previously observed.

The remainder of this paper is organized as follows. We first describe our simulation methodology, performance metrics, and application workload in section 2. Section 3 presents the miss rate and network traffic associated with each of our applications under the WI and WU coherence protocols. In section 4, we explore the effects of bandwidth and cache block size on the performance of the two protocols for each of our programs, using running time as our main evaluation metric. In section 5, we explain the causes of poor WU performance, introduce our categorization of update traffic, and evaluate several improvements to WU with respect to the categorization and execution time. Section 6 investigates the performance of the protocols we consider on next-generation architectures. Section 7 contains our conclusions.

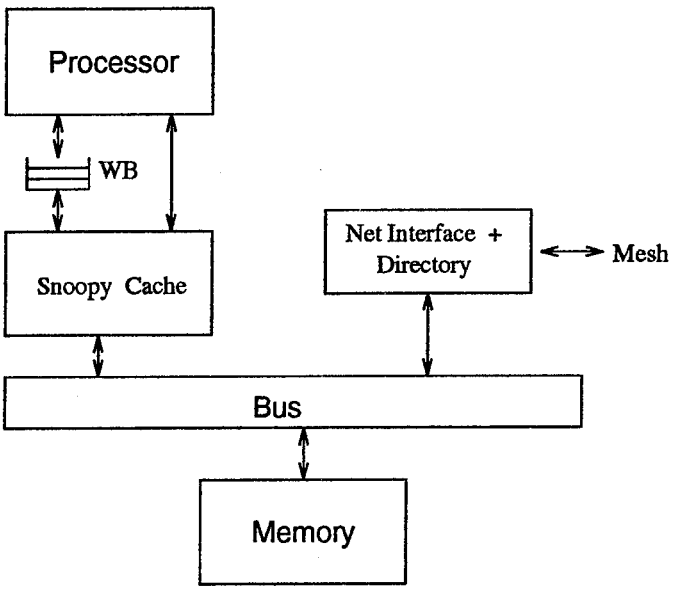

Figure 1: Node Architecture

2 Simulation Methodology and Workload

We are interested in exploring the relationship between cache coherence protocol, bandwidth, and cache block size in large-scale shared-memory multiprocessors, and therefore we use simulation for our studies.

2.1 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that exploits a mixture of interpretation and native execution to simulate unmodified MIPS R3000 object code efficiently. We simulate events at the level of processor cycles; all simulation parameters and results are expressed in terms of processor cycles.

We simulate a scalable direct-connected multiprocessor with 32 nodes. As seen in figure 1, each node in the simulated machine contains a single processor, a write buffer, cache memory, local memory, directory memory, and a network interface. Each processor has a 64 KB direct-mapped data cache. The data cache block size, the unit of fetching and coherence, is one of the parameters in our simulations; we consider small (16 bytes), medium (64 bytes), and large block sizes (256 bytes).

All instructions are assumed to take one cycle. A data read that hits in the cache also takes 1 cycle. Read misses stall the processor until the read request is satisfied. Writes go into an 8-entry write buffer and take 1 cycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. Furthermore, reads have priority over writes for accessing the cache and memory bus, but are prevented from accessing the cache while a write is updating it.

In order to model the limited number of pins in real processors, reads and writes contend for off-chip access. The cache is assumed lockup-free, so that the cache can be accessed while off-chip references are pending. Reads or writes may be locked out of the cache whenever the cache is being invalidated or updated from the outside. The lockout period is 2 cycles and occurs (if necessary) at the end of a memory bus operation.

A pipelined memory bus (clocked at one fourth of the speed of the processor) connects the main components of each machine node. A new bus operation can start every 20 processor cycles. The memory can provide the first word 32 processor cycles after the request is issued. The width of the memory bus is another parameter of our study and varies according to the network path width.

The simulator implements a full-map directory for controlling the state of each block of memory. Shared data are interleaved across the machine at the block level. Each node contains the directory for the memory associated with that node.

The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. The network clock speed is the same as the processor clock speed. Switch nodes introduce a 5-cycle delay to the header of each message. We experiment with two finite bandwidth networks with 16 and 64-bit wide data paths. In these networks (derived from the Alewife cycle-by-cycle network simulator), contention for network links and buffers is fully captured. For comparison purposes we also implement an idealized, infinite bandwidth network, in which the path width is always larger than the size of messages and contention is only modeled at each message's source and destination. Each network interface has a queue for out-going messages, which is fed either by the cache or the memory module at the node. In all our machine configurations the bus width is the same as the network link width.

Our WI protocol keeps caches coherent using the DASH protocol with release consistency [Lenoski *et al.*, 1990]. In our WU implementation, a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowledgement to the writing processor. Since we assume release consistency, the writing processor does not have to wait for update acknowledgements before continuing execution; the processor only stalls waiting for acknowledgements at a lock release point. Under this protocol, blocks are evicted from the cache only due to replacement.

Our WU implementation includes two optimizations. First, when the home node receives an update for a block that is only cached by the updating processor, the acknowledgement of the update instructs the processor to retain future updates since the data is effectively private. Second, when a parallel process is created by *fork*, we flush the cache of the parent's processor, which eliminates useless updates of data initialized by the parent but not subsequently needed by that processor.

2.2 Performance Metrics

For the most part, our focus is on the running time of the parallel section of the code and its components (busy time, read latency, write latency, and blocked time). However, we also consider read miss rates, network traffic, and the cumulative execution time across processors. We concentrate on read misses because we assume relatively deep write buffers and release consistency, which serve to hide the cost of writes. The read miss rate is computed solely with respect to shared references; that is, the read miss rate is defined as the total number of read misses on shared data divided by

Application	Shared Refs	Shared Reads (% of shared refs)	Shared Writes (% of shared refs)
Mp3d	5.1 M	60 %	40 %
Barnes-Hut	19.0 M	97 %	3 %
CG	6.8 M	98 %	2 %
Em3d	6.7 M	91 %	9 %
Blocked LU	47.3 M	89 %	11 %
SOR	20.7 M	85 %	15 %

Table 1: Memory reference characteristics.

the total number of reads to shared data. We classify cache misses using the algorithm described in [Dubois *et al.*, 1993] as extended in [Bianchini and Kontothanassis, 1994].

2.3 Workload

Our application workload consists of six parallel programs: Mp3d, Barnes-Hut, CG, Em3d, Blocked LU, and SOR. Mp3d is a wind-tunnel airflow simulation of 30000 particles for 5 steps. Barnes-Hut is an N-body application that simulates the evolution of 2K bodies under the influence of gravitational forces for 4 time steps. Mp3d and Barnes-Hut are part of the SPLASH suite [Singh *et al.*, 1992]. CG uses the conjugate gradient method to compute an approximation to the smallest eigenvalue of a 1400 $\times$ 1400, sparse, symmetric positive definite matrix with 78148 non-zero elements. Our CG implementation is a C version of the CG kernel from the NAS parallel benchmarks suite [Bailey et al., 1994]. Em3d simulates electromagnetic wave propagation through 3D objects. We simulate 20000 electric and magnetic nodes connected randomly, with a 10% probability that neighboring nodes reside in different processors. We simulate the interactions between nodes for 30 iterations. Blocked LU is an implementation of the blocked right-looking LU decomposition algorithm presented in [Dackland *et al.*, 1992] on a 384 $\times$ 384 matrix. SOR performs the successive over-relaxation of the temperature of a metal sheet represented by two 384 $\times$ 384 matrices. Table 1 summarizes the distribution of shared references in our applications.

3 Miss Rates and Message Traffic of WI vs. WU

In this section we examine the miss rates and network traffic produced by WU and WI protocols on our application suite, so as to quantify the lower miss rates of WU and reduced message traffic of WI.

Figure 2 presents the read miss rate of our applications for WI (left) and WU (right) (assuming 64-byte cache blocks). The percentage at the top of each column represents the percent of all read references to shared data that result in a miss; within a column misses are classified as either eviction, cold start, true sharing, or false sharing misses. Since WU only removes blocks from the cache due to evictions, WU has only eviction and cold-start misses; sharing-related misses are eliminated since

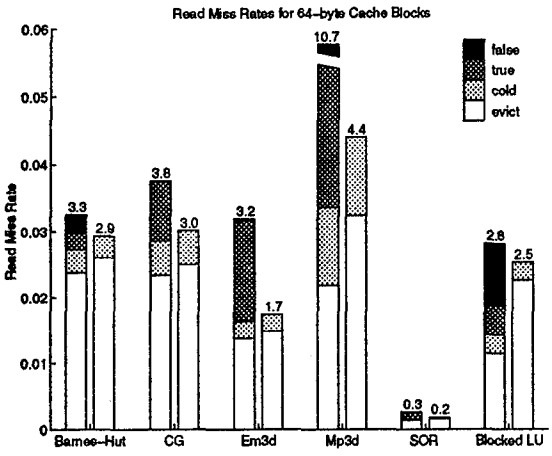

Figure 2: Miss rate under WI vs. WU.

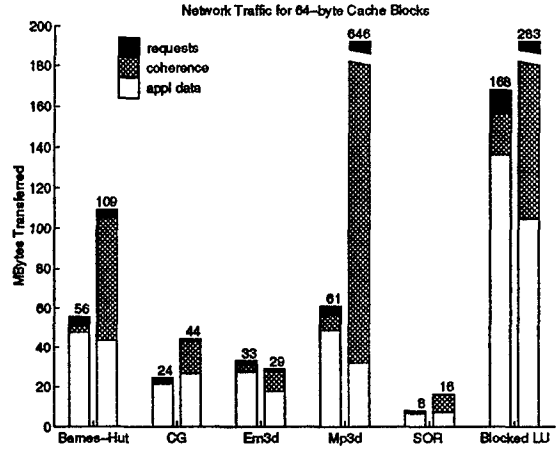

Figure 3: Bytes transferred under WI vs. WU.

multiple writes to the same block can be issued by one or more processors without causing any of them to lose that block.

From figure 2 we can see that, as expected, WU always results in lower read miss rates, even though WI usually exhibits lower replacement miss rates as invalidations effectively free up cache space. For `Mp3d`, `Em3d`, and `SOR` the difference in read miss rate is particularly large; the read miss rate under WI is a factor of 50-140% higher than under WU. For other applications, this difference is not as significant, as replacements dominate the miss rate under both protocols. These general effects are consistent across the block sizes we consider, although the magnitude of the miss rate differences varies slightly for other block sizes.

Although WU produces lower miss rates, it is at the cost of many update messages. Figure 3 presents the total number of bytes transferred by each coherence protocol for each application (again assuming 64-byte cache blocks). The number at the top of each column represents the number of MBytes transferred by the two protocols; within a column the traffic is classified as either data, coherence (includes invalidations, updates, and acknowledgements), and requests. This figure clearly shows that in terms of the number of bytes transferred, WU requires much more network traffic than WI, except in the case of `Em3d`. For `Mp3d` there is more than an order of magnitude difference in the amount of data transferred by the network, 646 Mbytes for WU vs. 61 Mbytes for WI. In terms of the number of messages sent, this corresponds to 70 M messages for WU vs. 2.5 M messages for WI. `Barnes-Hut` and `SOR` have lower miss rates, and therefore require less communication, but the difference between WU and WI is again substantial; WU transfers nearly twice as many bytes as WI, and requires about 5 times as many messages. Again, this effect is consistent across block sizes.

`Em3d` is an exception as the network transfers less data under WU than under WI, independently of the cache block size. The most important reason for this effect is that the increase in coherence traffic associated with WU is somewhat limited, since cache blocks effectively shared are very infrequently written in this program.

Comparing the amount of data and coherence traffic involved in our applications (shown in figure 3), we observe that the network traffic associated with WU is dominated by updates and their

corresponding acknowledgements (rather than misses) in most cases, and this traffic usually grows significantly with an increase in block size. Even if a larger block size produces a lower miss rate, and hence less network traffic due to misses, the reduction in network traffic due to fewer misses is overwhelmed by an increase in the number of updates.

In summary, WU can lower the miss rate by roughly 10-60% over WI, while increasing network traffic by as much as an order of magnitude. The benefits of the lower miss rate depend on the remote access latency of the machine, while the costs of the additional network traffic depend on the available network bandwidth. In the next section, we consider whether future increases in network bandwidth will be sufficient to resolve this tradeoff in favor of WU.

4 The Effect of Bandwidth and Block Size on Performance

In this section we consider whether expected increases in network bandwidth enable WU to outperform WI on a scalable machine. We explore the effect of network (and memory) bandwidth on the running time of our applications under both WI and WU. We also investigate how changes in block size affect our comparison of the protocols.

Figures 4-9 present the running time of each application in our suite under the two different protocols, for three levels of bandwidth and a range of cache block sizes. As seen in the figures, WI performs better than WU for `Barnes-Hut` and `Mp3d`; the two protocols achieve comparable performance for `CG` and `SOR`; and WU outperforms WI for `Em3d` and `Blocked LU`.

In general, any comparison between WI and WU protocols must consider the negative impact of the higher cost of read accesses under a WI protocol, in contrast with the potential degradation caused by the excessive network traffic generated under a WU protocol. Figures 4-9 illustrate the effect of bandwidth and cache block size on this tradeoff. Medium bandwidth significantly degrades the performance of 256-byte blocks (which produce the lowest miss rates for all applications and protocols, except `Barnes-Hut`) regardless of the protocol.

Increasing the bandwidth alleviates the performance degradation associated with the larger blocks, while not significantly affecting the performance of the smaller blocks in most cases. Extremely high bandwidth should provide a clear performance advantage to the WU protocol independently of the block size, since the update traffic should have very limited performance implications and the cost of reads would still be higher under the WI protocol as a result of its higher miss rates.

This is not the effect we observe in our simulations, however. Given infinite bandwidth, WU performs slightly better (if at all) than WI for small cache blocks, while the performance of the two protocols with large blocks is indistinguishable for three of our applications. The exception is `Mp3d`, for which WU is significantly worse across all block sizes, even though WU results in lower miss rates. The fact that infinitely wide data paths do not enable WU to perform much better than WI is somewhat surprising, and it suggests that bandwidth alone is not enough to justify using WU.

The enormous network traffic associated with WU causes several forms of performance degradation ultimately due to an increase in network and memory congestion. The degradation may manifest itself in many different ways, such as (1) increased read latency, (2) increased write latency induced by processor stalls due to full write buffers, (3) increased synchronization overhead whenever processors must wait for update acknowledgements before releasing locks, and (4) increased synchronization latencies whenever the processor holding a lock has its (blocking) reads delayed by previous

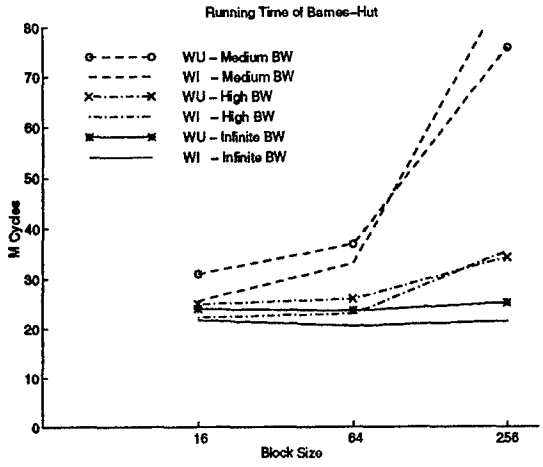

Figure 4: Running Time of Barnes-Hut.

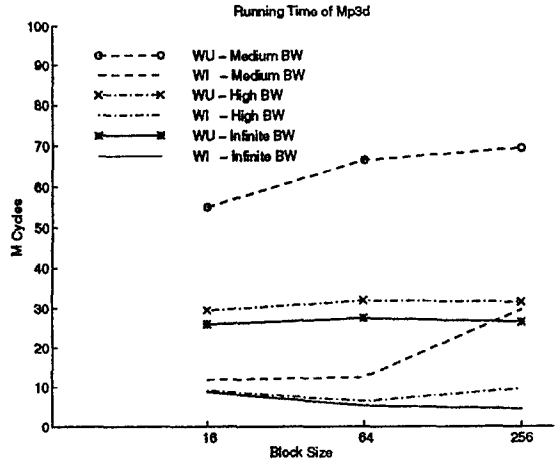

Figure 5: Running Time of Mp3d.

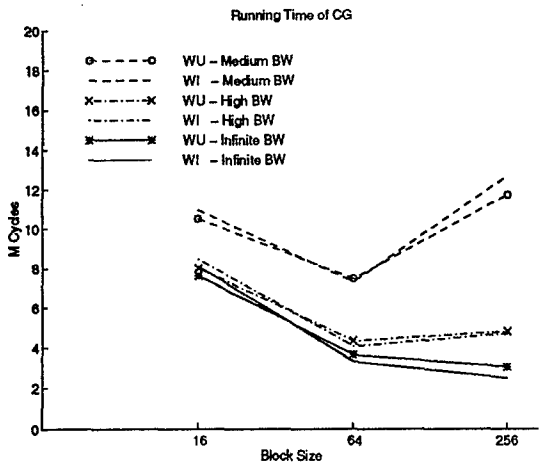

Figure 6: Running Time of CG.

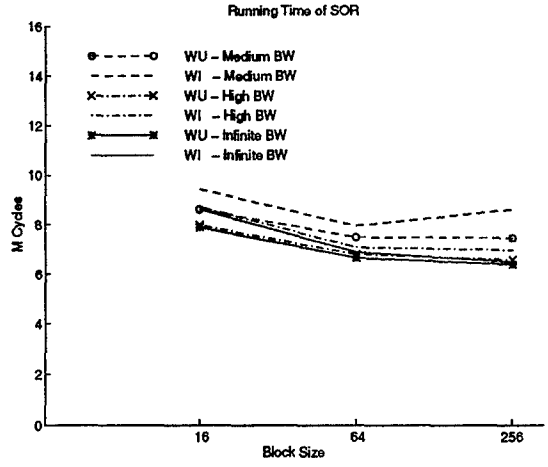

Figure 7: Running Time of SOR.

messages. In addition, an excessive number of update messages may cause processors to be locked out of their caches frequently.

We can observe the contribution to running time of these effects in figures 10-15. These figures break down the cumulative running time of processors (normalized to the time for WI with 16-byte blocks) under infinite bandwidth. The WI performance is shown on the left of each pair of bars; WU is on the right. The categories of time are (from top to bottom): processor blocked (or, in other words, lock acquire) overhead, lock release (including write buffer flush) overhead, stall time due to a full write buffer, stall time when a processor cannot access its cache (lockout), read access cost minus the lockout time, and processor busy time.

These figures show that each effect described above contributes to the tradeoff between WU and WI. Barnes-Hut (figure 10) performs better under WI than WU due to the relatively high (and constant) cost of read accesses, and the extra lock release and processor blocked overheads under WU. For Mp3d, read latencies and processor blocked overheads are extremely high for WU for all

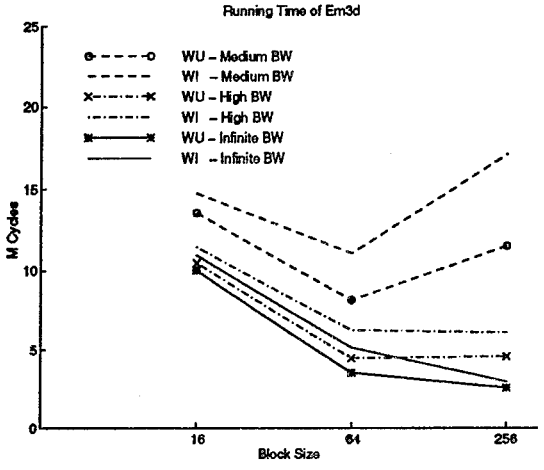

Figure 8: Running Time of Em3d.

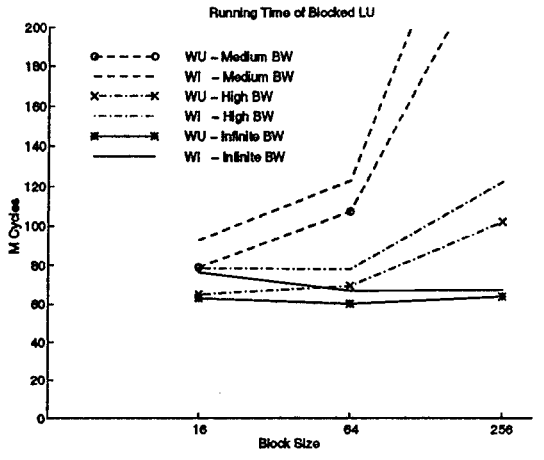

Figure 9: Running Time of Blocked LU.

block sizes, and dominate the comparison against WI. Lock release overheads hurt WU performance for CG (figure 12), especially with the larger block sizes. The performance of the two protocols is comparable in most cases for SOR. For Em3d, WU performs better than WI, especially with 64-byte blocks; the performance of WU with 256-byte blocks is degraded by lockout overhead. For Blocked LU, the read latency under WU is slightly lower than under WI for 16 and 64-byte blocks. With 256-byte blocks the latency of reads under WU is actually larger than under WI, even though the WU miss rate is lower. The performance advantage obtained by WU is due to significantly lower processor blocked overheads than under WI. The reason for this effect is that Blocked LU is plagued by a large sequential processing component, which is reduced as a result of the fewer misses under a WU protocol.

It is clear from these figures that a lack of bandwidth is not the only problem with WU. Figures 4-7 showed that no amount of bandwidth enables WU to perform significantly better than WI in all cases. Although hardware techniques designed to alleviate a particular source of overhead, such as cache lockout, might help, the most significant source of overhead is the update messages themselves. We will now consider how to reduce the network traffic associated with WU.

5 Improving Write Update Performance

The previous section showed that high bandwidth is not enough to enable WU to consistently outperform WI. The excessive number of updates produced by WU introduces several performance problems, all of which could be alleviated by reducing the number of messages used by the WU protocol. We first consider how many updates are actually required for correct execution of the program, and then evaluate techniques for combining multiple updates in one message and eliminating useless updates.

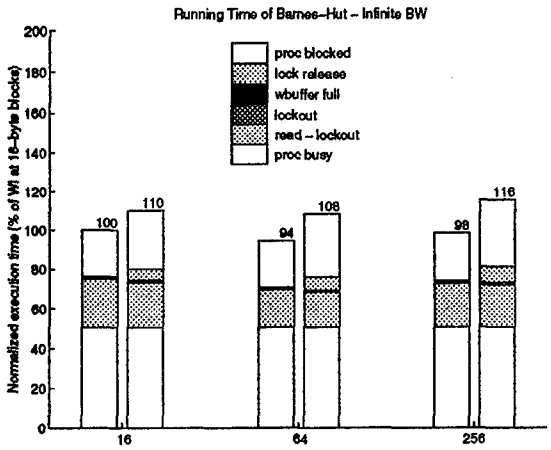

Figure 10: Cumulative Running Time of `Barnes-Hut`.

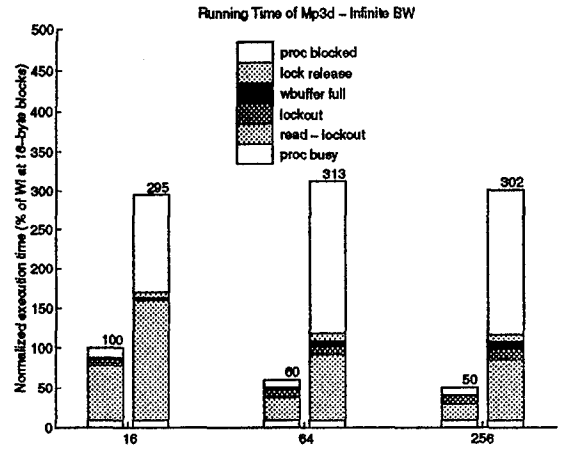

Figure 11: Cumulative Running Time of `Mp3d`.

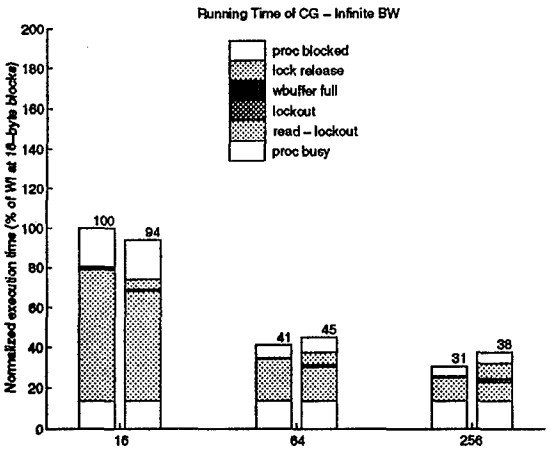

Figure 12: Cumulative Running Time of `CG`.

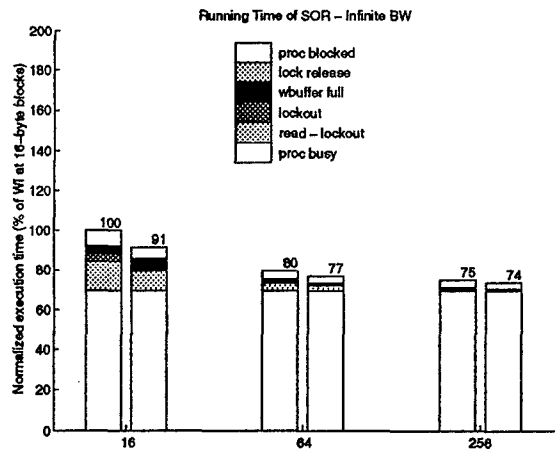

Figure 13: Cumulative Running Time of `SOR`.

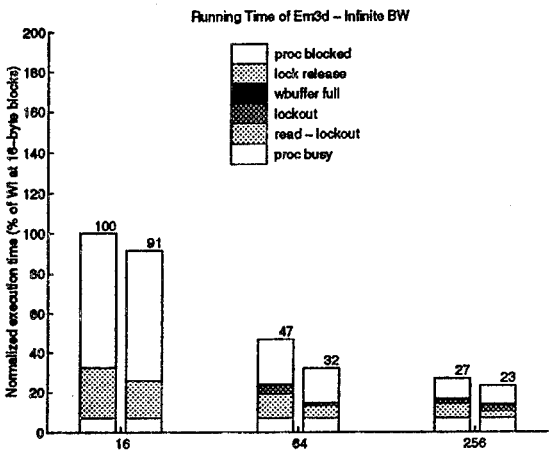

Figure 14: Cumulative Running Time of Em3d.

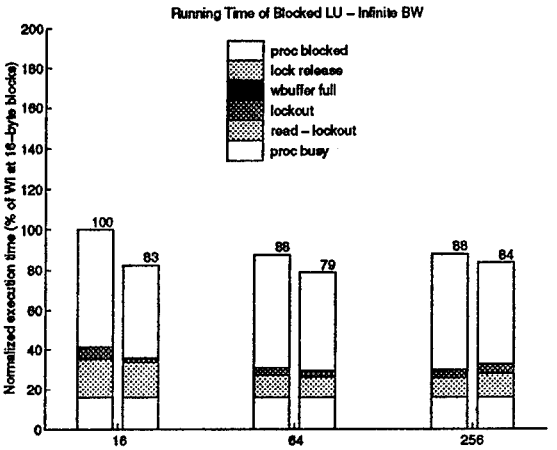

Figure 15: Cumulative Running Time of Blocked LU.

5.1 Useful vs. Useless Updates

In order to investigate ways in which to eliminate update messages, we classify updates in terms of their usefulness to the processors receiving them. We classify updates as *useful* and *useless*. An update is useful if the processor references the updated word before the next update of that word arrives; otherwise the update is useless. Intuitively, useful updates are those updates required for correct execution of the program, while useless updates could be eliminated entirely and not affect the correctness of the execution.

We divide useless updates into three categories: *proliferation*, *false*, and *termination* updates. Update messages are classified at the end of an update's lifetime, which happens when it is overwritten by another update to the same word or when the program ends. If, between two updates to a word, the cache block containing the word is not referenced, then we classify the first update as a proliferation update. If another word in the same block is referenced, then we classify the first update as a false (sharing) update. If a processor receives an update and the program terminates without referencing the block again, then we classify the update as a termination update.

This categorization is fairly straightforward, except for our false update class. Successive (useless) updates to the same word in a block are classified as proliferation instead of false sharing updates, if the receiving processor is not concurrently accessing other words in the block. Thus, our algorithm classifies useless updates as proliferation updates, unless *active* false sharing is detected or the application terminates execution.

Although by no means unique, our categorization is intuitive, while being sufficiently simple to compute, as it does not require any future knowledge of sharing behavior or an excessively large amount of memory. Greater details about the categorization and the algorithms we use in our simulations can be found in [Bianchini and Kontothanassis, 1994].

An analysis of the update messages sent during our simulation experiments shows that the number of useless updates is extremely high: more than 90% of all updates sent during execution of Barnes-Hut, Mp3d, and Blocked LU are useless, while between 55% and 90% of all updates in CG,

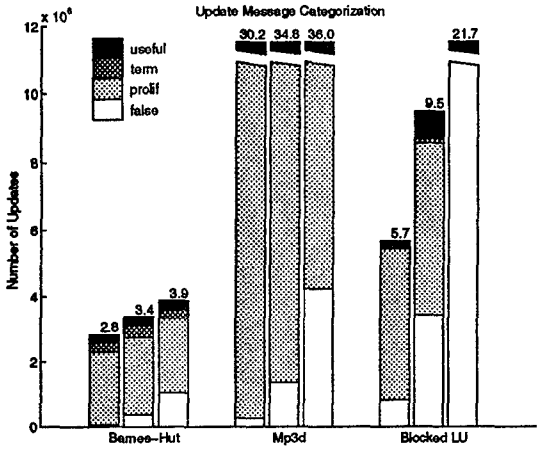

Figure 16: Categorization of Updates for Group 1.

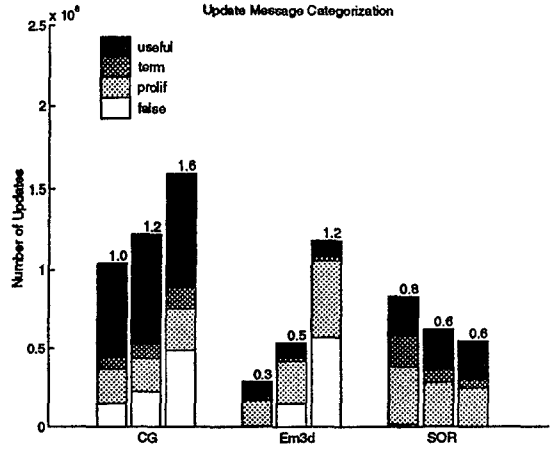

Figure 17: Categorization of Updates for Group 2.

Em3d, and SOR are useless. Despite the two optimizations described in section 2, which eliminate useless updates for data that is effectively private and for data that is initialized by a parent process prior to a *fork*, the vast majority of the remaining updates are still useless.

Figures 16 and 17 present the number and types of useless updates found in our programs. We separate the applications into two groups according to the percentage of useless updates found in the applications. Applications with more than 90% useless updates consistently across block sizes are in the first group. The other applications are placed in the second group. For each application, the columns from left to right correspond to 16, 64, and 256-byte blocks, respectively. The number on top of each column represents the total number of updates (in millions).

Proliferation updates clearly dominate in two of the programs, Barnes-Hut and Mp3d. Proliferation updates are also an important factor for Blocked LU and Em3d, but false updates dominate when using the largest block size we considered. Useful updates are more numerous in CG, while proliferation and useful updates are major contributors in SOR. In most cases, increasing the block size causes a substantial increase in the number of proliferation updates due to the fact that, with larger blocks, sharing becomes more widespread and processors rarely drop copies of data they no longer need. The only exception to this trend is SOR, which exhibits a slight reduction in the number of updates as we increase the block size. This result is simply a side-effect of the different timings associated with the different block sizes. Recall that our WU protocol does not issue updates (after the first one, of course) if no other processor is sharing the written block. Thus, in the beginning of the computation, as the block size is increased, processors have more time to write to cache blocks not yet shared by other processors.

By relating the frequency of useless updates back to the source code, we can gain insight into the sharing patterns that produce them. Our analysis shows that widely-shared data are one common source of useless updates. A single processor that modifies data in a critical section sends an update message to each processor that has ever accessed the variables in the critical section, even though only one processor (the next one to enter the critical section) will need those updates. Shared counters and global work queues both produce numerous useless updates of this type. For example, 98% of

the updates (with 64-byte cache blocks) in Mp3d can be found in a single routine that updates the cell array describing the state of the simulation space. More than 90% of those updates are proliferation updates. Barnes-Hut also uses a global counter, which accounts for 4-5% of the proliferation updates depending on the block size.

Multiple consecutive writes to the same word may also cause a large number of proliferation updates, as each write except for the last one is guaranteed to be useless to the processors sharing the cache block. The sequential LU phase of Blocked LU generates useless updates due to this referencing behavior.

Under certain circumstances pair-wise sharing can be an important source of useless updates. If two processors share data and neither is the home node for the data, then each write to the shared data results in a useless (proliferation) update to the home node. Thus, even if the data is truly shared between the two processors, there will be a proliferation update per useful update. Consider SOR as an example. Of the 280K proliferation updates in the program, 260K updates (which is over 40% of all updates) are proliferation updates to the home node. These proliferation updates could be eliminated by ensuring that every boundary row in the matrix has one of the two processors sharing that row as the home node.

Load balancing schemes are another common source of useless updates. Work that migrates to an idle processor may leave behind a copy of the data in the cache, causing updates to retrace the path of migration. Roughly 46% of the updates in Barnes-Hut with 64-byte blocks occur in a single routine, and the vast majority of those updates are useless, proliferation updates caused by moving a body from one processor to another without flushing the cache. The more load imbalance occurs, the more bodies move among processors, creating new recipients for updates.

False sharing is another source of useless updates. For Blocked LU with 64-byte cache blocks, 40% of all updates are false updates; the percentage of false updates rises to 72% with an increase in block size to 256 bytes. CG and Em3d also suffer from severe cases of false updates when using 256-byte blocks.

It is clear from these figures that the elimination of useless updates would improve WU performance tremendously. In some cases eliminating useless updates may be easy; our examination of the useless updates to the cell array in Mp3d uncovered the fact that certain values in the cell array were repeatedly modified but never used.[1] By eliminating the useless code, we improved WI and WU running time performance by factors of 2 and 9, respectively, as well as reduced the number of updates by a factor of 2.5-3 depending on block size. Nonetheless, over 85% of all updates in the modified program are still useless, proliferation updates. In our subsequent experiments we will use this modified Mp3d, referred to as New Mp3d, in place of the original.

The following sections describe and evaluate more complicated techniques for reducing the number of useless updates. We first evaluate coalescing write buffers as a mechanism for combining multiple updates in a single message, and then consider how to eliminate false and proliferation updates.

[1]The version of Mp3d in the SPLASH suite may have been part of a larger program that used these values for simulation statistics. Our analysis of updates uncovered the fact that the source code distributed in the SPLASH suite contains this useless code.

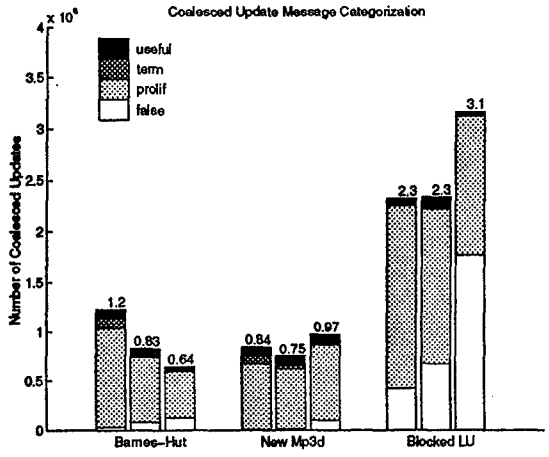

Figure 18: Categorization of Coalesced Updates for Group 1.

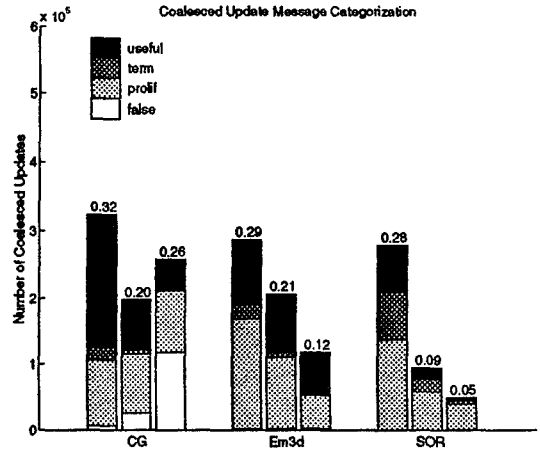

Figure 19: Categorization of Coalesced Updates for Group 2.

5.2 Merging Updates with Coalescing Write Buffers

A coalescing write buffer [Jouppi, 1993; Thacker *et al.*, 1992] is simply a cache-block-wide buffer capable of merging writes to the same cache block. In the context of a WU protocol, this feature allows for a reduction in the number of updates propagated outside the processor. A coalescing write buffer is also useful for WI, since it usually reduces the average number of occupied buffer entries, and therefore induces fewer processor stalls. Note that a coalescing write buffer is slightly different from a write cache (e.g. [Dahlgren and Stenstrom, 1994]). A write cache sits between the cache and the memory bus and therefore has no effect on the write traffic going from the processor to the cache.

Our implementation of coalescing write buffers assumes 4 entries, each wide enough for a cache block.[2] We associate a dirty bit vector with each entry in a write buffer indicating the words that were written. If the processor writes to an address in a cache block that is already in the buffer, the new write is merged with earlier ones, and its dirty bit is set. If the processor writes to a block for the first time, a new buffer entry is allocated for the write. Unlike traditional write buffers, our coalescing buffer does not attempt to write its entries out immediately; it waits until there are 2 valid entries in the buffer or until it is forced to flush entries. When a write is issued from the buffer, only the dirty words are sent in the message. A coalesced update message locks out the cache of the receiver for the same number of cycles as it takes to transfer an entire cache block on the bus, subject to the constraint that the lockout time has to be at least the same as the number of dirty words in the coalesced message.

Our experimental results show that the addition of coalescing write buffers improves the performance of both protocols; WI improves slightly, while WU improves significantly. The coalescing buffer's ability to combine multiple writes to a specific word does not provide significant performance gains for our applications, however. The only application that benefits from this characteristic of the coalescing buffers is `Barnes-Hut`. Under the WU protocol, this application achieves a 5-28%

[2] In the previous experiments we assumed an 8-entry non-coalescing write buffer. We reduce the number of entries in the coalescing buffers because each entry takes up more chip area under coalescing.

reduction in the total number of word updates sent across the network, depending on the block size.

Coalescing reduces the number of bytes transferred under WU by at least 20% in most cases. This reduction comes mainly from requiring fewer message headers for the coherence traffic.

The most significant gains provided by coalescing write buffers come from a major reduction in the number of coherence messages transferred through the network, with a corresponding reduction in the number of acknowledgements required. Figures 18 and 19 depict our categorization of the coalesced update transactions involved in our two groups of applications. Again, for each application, the columns correspond to 16, 64, and 256-byte blocks from left to right. The number on top of each column is the total number of coalesced update messages (in millions).

Our categorization of coalesced messages is a slight modification of the one for updates with traditional write buffers. A message is considered useful (true sharing) if at least one of the updates included in the message is useful. A false sharing message is one in which none of the updates is a true sharing update and at least one of the updates is a false sharing update. A message is classified as a proliferation message if all of the updates in the message are proliferation updates. Proliferation messages at the end of the program are classified separately as termination messages.

Comparing these figures with the ones in the previous section, we can see that, for all our programs, coalescing write buffers reduce the number of coherence messages under WU by at least 60%, except for Em3d with 16-byte blocks. All applications achieve more than 70% improvement in the number of coherence messages with the largest cache block we consider. SOR achieves the greatest reduction in the number of messages, 91% with 256-byte blocks, due to its perfect spatial locality.

We can also gather from the figures that coalescing changed the overall update behavior of two applications completely (Barnes-Hut and Em3d); for these applications, increasing the block size results in a reduction of the total number of update messages. This shows that these applications exhibit good spatial locality of writes to cache blocks. Three other applications (New Mp3d, Blocked LU and CG) exhibit good spatial locality of writes up to 64-byte blocks; larger blocks reverse the trend of decreased coherence traffic by significantly increasing the degree of sharing in the programs.

In terms of the percentage of useful update traffic with and without coalescing, our figures depict mixed results. Barnes-Hut and New Mp3d exhibit roughly the same percentage of useful traffic, independently of whether coalescing is used. Em3d exhibits an increase in the percentage of useful coherence traffic with coalescing and the larger cache blocks, while the other three applications suffer a severe decrease in the percentage of useful traffic. These results show that, although coalescing significantly reduces the number of messages and bytes sent across the network, there is plenty of room for further improvement as most of the coherence traffic is still useless.

As examples of the performance improvement provided by coalescing, compare figures 20 and 4. Figure 4 shows the running time of Barnes-Hut with conventional write buffers as a function of bandwidth and block size, while figure 20 shows the same type of graph but assumes coalescing write buffers. Figure 21 shows the coalescing running times for New Mp3d. Coalescing provides running time improvements under WU of as much as 43% for New Mp3d with infinite bandwidth and 256-byte blocks, and 19% for Barnes-Hut also with infinite bandwidth and 256-byte blocks. Other applications, such as CG and Blocked LU, also exhibit significant improvements in running time under WU and wide coalescing buffers with infinite bandwidth.

Performance improvements quickly degrade as we decrease the bandwidth available in the system, however. For instance, the performance of New Mp3d under WU with coalescing, high bandwidth, and

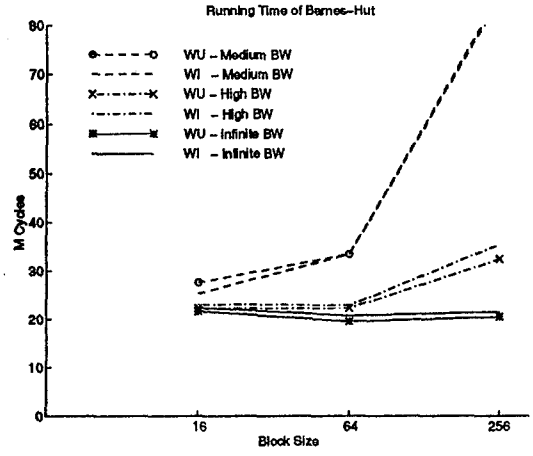

Figure 20: Running Time of Coalescing `Barnes-Hut`.

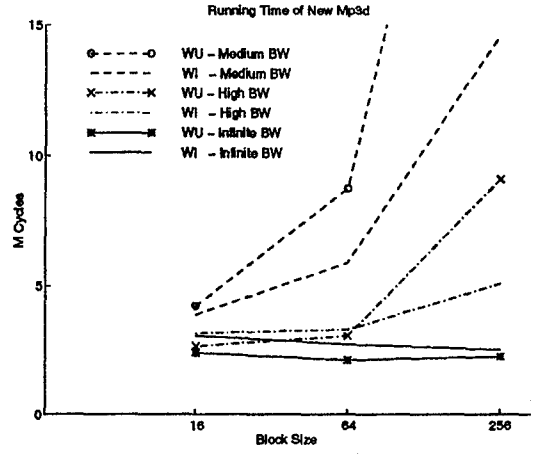

Figure 21: Running Time of Coalescing `New Mp3d`.

256-byte blocks is a factor of 2 worse than with traditional write buffers. The reason for this effect is that, in the presence of a non-uniform distribution of memory accesses, coalescing write buffers may cause severe memory and network contention, as a result of the longer period of time resources remain busy per request. Larger cache blocks (and, therefore, potentially longer transactions) and lower bandwidth make this scenario worse. Shorter requests result in a greater degree of interleaving in network and memory utilization, allowing more processors to continually make forward progress. `New Mp3d` is again a good example. Under high bandwidth and 16-byte blocks, the performance of the program is 18% better with coalescing buffers than without them.

In short, when coalescing write buffers are effective at merging updates to large cache blocks, the result is long messages that occupy the network and memory for long periods of time and may cause contention. When coalescing write buffers are not effective at merging updates, wide buffers designed to hold large cache blocks waste chip space. These observations indicate that wide coalescing write buffers are not necessarily profitable.

5.3 Eliminating False Updates

Assuming the largest cache block we consider, most of the updates in `Blocked LU` are false updates. We can eliminate many of these updates by reducing false sharing in the program. Several techniques have been proposed for reducing false sharing misses under WI; these techniques would also serve to reduce false updates in WU. They include padding data items to the size of a cache line [Torellas *et al.*, 1990], aligning data structures on cache line boundaries, and reorganizing data structures so as to combine data items with similar sharing patterns [Eggers and Jeremiassen, 1991]. We experiment with two other techniques, *software caching* [Bianchini and LeBlanc, 1992] and *indirection* [Eggers and Jeremiassen, 1991], to reduce false sharing in `Blocked LU`. We call the resulting programs `SC Blocked LU` and `Indirect Blocked LU`, respectively.

Software caching consists basically of copying a range of virtual addresses to a different range of virtual addresses, allowing the application to determine when data copies are made, when local

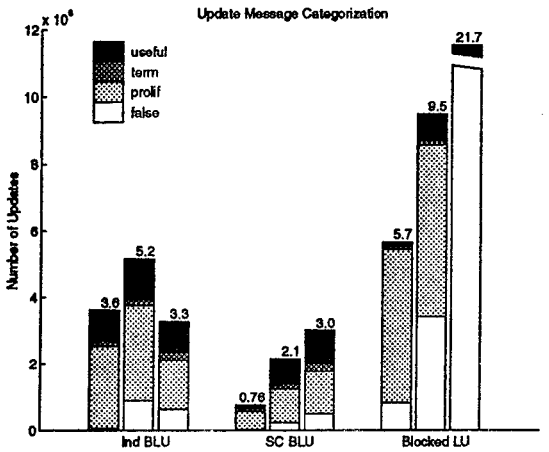

Figure 22: Categorization of Updates for Different Versions of `Blocked LU`.

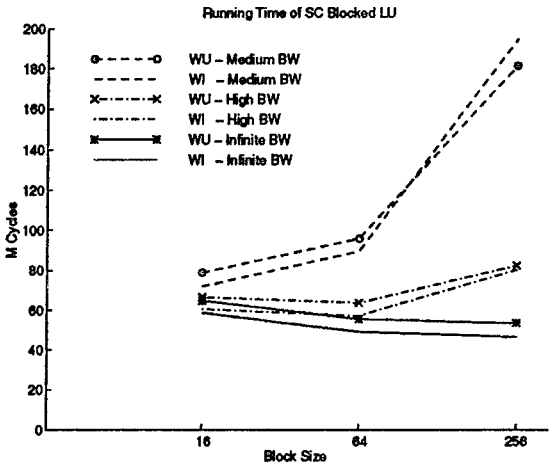

Figure 23: Running Time of `SC Blocked LU`.

data is written back to the global data space, and how the data copies are organized. As a result, the coherence unit and the coherence protocol are both defined by the application. An application can tailor the unit of coherency to the data, thereby avoiding false sharing. It can also "schedule" writes to the global data space in order to alleviate contention. Finally, the application can change the data layout in order to reduce the number of replacement misses.

In `SC Blocked LU`, each processor makes a local, re-organized, copy of the data it needs, modifies the data as appropriate, and copies the result back to the original location when required for data sharing. Copy backs are scheduled in such a way that congestion in the network and memories is relieved. This technique incurs the overhead of making a local copy, but eliminates coherence transactions caused by false sharing and reduces the number of replacement misses in the program. (Note that some false sharing may occur when the data is copied back to its initial location, but this overhead is unavoidable if the algorithm requires the modified data back in the original location.)

Indirection changes a vector of data items to a vector of pointers to blocks of data items, effectively reorganizing the allocation of data to cache lines and thereby eliminating false sharing. Indirection also introduces the overhead of an indirect reference on each access to shared data. Thus, this technique presents a tradeoff between the reduced number of misses (or updates) versus the additional computation associated with references.

Our simulation results show that software caching does reduce the network traffic of WU substantially. `SC Blocked LU` with 256-byte cache blocks sends about 3M update messages and 149MB of data through the network, while `Blocked LU` sends about 22M updates and 680MB of data with the same block size. Note that, for these cache block sizes, software caching reduces the number of bytes sent through the network substantially more than coalescing does, 78% against 33%.

As seen in figure 22, the reduction in the number of update messages sent by `SC Blocked LU` across the network comes from a 97% reduction in the number of false sharing updates, and a 77% reduction in proliferation updates. The percentage of useful updates increased from 4% to 33% of the total number of updates, which is still somewhat small.

Figure 22 shows that indirection also reduces the network traffic of WU substantially. `Indirect`

`Blocked LU` with 256-byte cache blocks sends 3.3M updates and 507MB of data through the network. As in the case of software caching, these improvements come from major reductions in the number of false and proliferation updates. As a result, the percentage of useful updates goes up to 29%.

`SC Blocked LU` exhibits much better running time performance (shown in figure 23) than `Blocked LU` for the larger cache blocks, while `Ind Blocked LU` (not shown) is not as successful. The latter program suffers from the overhead of indirection and an increase in the number of misses taken by the one processor responsible for the sequential LU portion of the algorithm, and this processor is on the critical path of the computation.

Comparing the WI and WU executions of these programs, we see that the impact of our program modifications is greater for WI than for WU. The WU implementations suffer greatly from an increased critical path of execution, in the case of software caching, due to excessive update traffic backing up at the processor running the sequential LU phases of the algorithm. In the case of indirection, both the update traffic and an increased replacement miss rate are the performance problems. In section 5.5, we investigate whether coalescing can alleviate these problems.

5.4 Eliminating Proliferation Updates

In this section, we evaluate the performance of two strategies that can use invalidations to reduce the number of useless updates in update-based protocols. The first strategy consists of a hybrid WI/WU coherence protocol that allows for the static determination of the protocol to use on a per cache block basis. This strategy is an extension of the work presented in [Veenstra and Fowler, 1992] and is referred to as our static hybrid protocol. The second strategy we study can also be considered a hybrid protocol in which processors dynamically self-invalidate cache blocks according to the update traffic directed to them. Protocols of this kind are usually referred to as *competitive update* protocols [Karlin *et al.*, 1988].

Static Hybrid

A static hybrid protocol must decide, for each cache block, whether that cache block should be managed with WU or WI. A *selection policy* is a rule for deciding whether a given cache block should use WU or WI. A block that uses WU is called a "WU-block" and a block that uses WI is called a "WI-block".

The motivation for using WU as opposed to WI for a certain cache block is that the read latency for the block may be reduced under WU, but only at the cost of some extra coherence traffic. Our selection policies estimate the read latency and the coherence overhead associated with each block under the different protocols. If a cache block has less coherence overhead with WU than read latency with WI, then that block should be included in the set of WU-blocks. In addition, it may be worthwhile to include a cache block in the set of WU-blocks even though that block has higher overhead with WU than with WI. The reason for this is that the updates can often be done in parallel with processor busy cycles, so the cost of the updates is hidden. Read misses, however, stall the processor. If too many cache blocks are included in the set of WU-blocks, however, then the additional update operations will increase network and memory contention. Thus, when deciding whether to include a cache block in the set of WU-blocks, the static hybrid selection policy must balance the potential reduction in cache misses against the potential increased cost of a cache miss.

The information needed by the selection policies is obtained from a simulation run that collects statistics about each cache block. The statistics needed to facilitate the selection of the WU-blocks are listed below.

- Rereads. The number of rereads for a cache block is the number of times a processor had to reread that block. If cache replacement effects are ignored, then using WU on a given cache block can eliminate all the rereads for that block.
- Extra updates. The number of extra updates is estimated by counting the total number of writes to a cache block and subtracting the number of invalidations required by WI for that cache block. Writes to a cache block that has not yet been shared are not included in the number of extra updates. This metric is intended to quantify the number of extra write operations that would be required if WU were to be used on that cache block.

The static protocol policies differ in how they use this information to select the WU-blocks. The first policy we study is conservative in selecting WU-blocks in that only those cache blocks that are estimated to have less coherence overhead using WU than read latency under WI are included in the set of WU-blocks. These blocks can be characterized by satisfying the following formula: (extra updates $\times$ cost of an update) $\leq$ (rereads $\times$ cost of a read). We will refer to the left and right-hand sides of this formula as WU-cost and WI-cost, respectively. We will refer to the sum of all WU and WI-costs of the blocks selected as Total WU-cost and Total WI-cost, respectively.

To the set of blocks chosen with our first selection policy, one can add blocks for which WU-cost > WI-cost. In this case, the rereads required by WI would be traded for the extra updates required by WU. Our second static hybrid policy selects blocks among the ones with the largest ratios (WI-cost / WU-cost) for inclusion into our first set of blocks. The policy includes just enough blocks to make Total WU-cost == Total WI-cost. More aggressive policies can be defined by allowing the number of WU-blocks to grow until the ratio between the two total costs exceeds a certain threshold. We study policies for which Total WU-cost is 10%, 20%, and 50% more than Total WI-cost. Our write stall statistics show that 25% of the non-overlapped cost of an update is a reasonable assumption for the update cost as observed by processors. We include more than one selection policy in our study in order to find a close approximation to the best policy.

Our experiments show that the second selection policy tends to deliver the best results overall for the static hybrid protocol. The effect of the protocol using this policy on the update traffic can be seen in figures 24 and 25. Comparing these results with the ones for our pure WU protocol (figures 16 and 17), we see that the static hybrid strategy is extremely successful at reducing the amount of (useless) coherence traffic in all applications, except for `CG`. This reduction in the amount of useless traffic does not significantly increase the percentage of useful updates however, since the static hybrid protocol eliminates both useless and useful updates, but does not guarantee that more useless than useful updates are prevented.

Comparing the coherence traffic results obtained for `SC Blocked LU` and `Blocked LU` under the static hybrid protocol, we see that software caching entails many fewer useless update transactions. Coalescing also compares favorably against the static hybrid technique; the former strategy generates as much as 38% less overall traffic than the latter for all applications and large block sizes.

The static hybrid protocol decreases the amount of coherence traffic associated with a pure WU strategy by simply not using updates for certain cache blocks, which may cause an increase in the

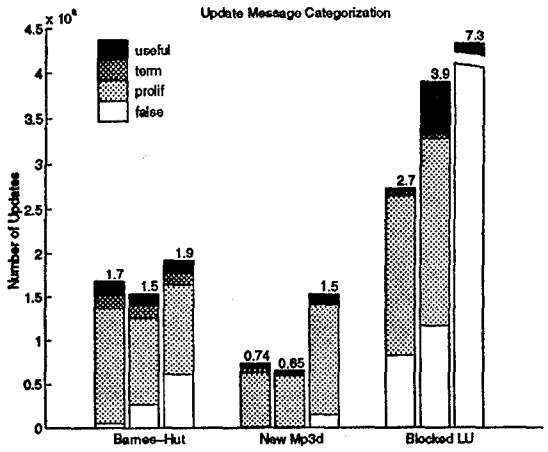

Figure 24: Categorization of Updates for Group 1 under Static Hybrid Protocol.

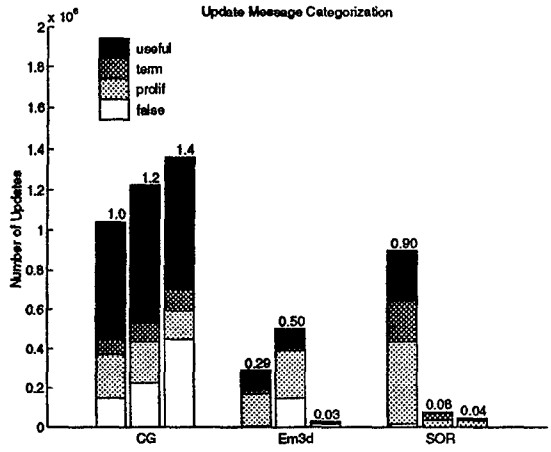

Figure 25: Categorization of Updates for Group 2 under Static Hybrid Protocol.

miss rate. Figure 26 presents the read miss rates of our applications under WI (left) and the static hybrid strategy (right). Note that, in cases where this miss rate degradation is significant (such as `New Mp3d` and `Blocked LU`), running time performance suffers accordingly. However, even when the miss rates do not increase noticeably (as for `CG` and `Em3d`), programs may perform worse as a result of not using WU for performance-critical blocks. `Barnes-Hut` was the only application to achieve execution time improvements under the static hybrid strategy for relatively high bandwidths. Under high and infinite bandwidths and 64-byte blocks, the static hybrid approach improves performance by 10% in comparison to the pure WU protocol. The performance of `Barnes-Hut` remains worse than WI under the static hybrid protocol, however.

Our experience with the static hybrid strategy clearly demonstrates that its performance is heavily dependent on good initial estimates for the cost of updates. The problem is that it is very difficult to produce cost estimates that can effectively be used for all applications. These results suggest that the static hybrid protocol is of limited use, unless compilers can produce accurate estimates of the cost of updates for each application.

Dynamic Hybrid

The dynamic hybrid strategy is inspired by the coherence protocols of the bus-based multiprocessors using the DEC Alpha AXP21064 [Thacker *et al.*, 1992]. In these multiprocessors, each node makes a local decision to invalidate or update a cache block when it sees an update transaction on the bus. The decision depends on the presence of the block in the primary cache. (The contents of the secondary cache are a superset of the contents of the primary cache.) When the block is present in the primary cache, the cache controller updates the copy in the secondary cache and invalidates the copy in the primary cache. If the block is updated again before any reference by the processor, the cache controller invalidates the copy in the secondary cache. Thus, after at most two update transactions, an unused cache block is invalidated from the processing node.

Our implementation of this idea associates a counter with each cache block and invalidates the

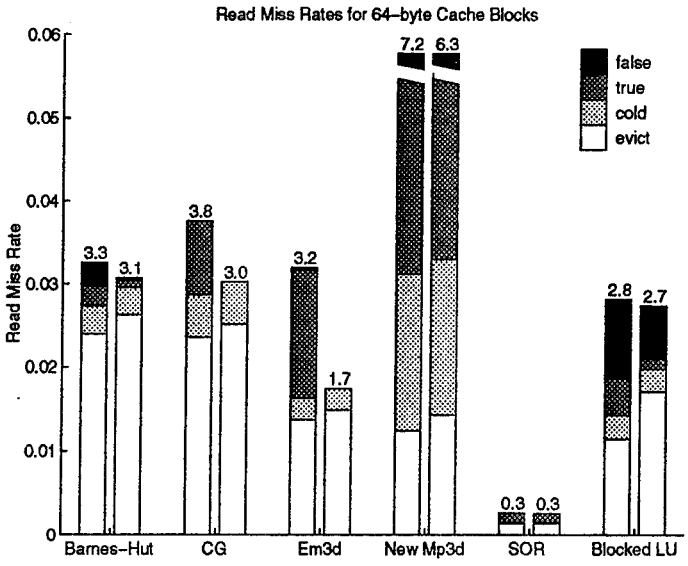

Figure 26: Miss Rate Under the Static Hybrid Protocol.

block when the counter reaches a threshold.[3] At that point, the node sends a message to the block's home node asking it not to send any more updates to the node. References to a cache block reset the counter to zero. We use counters with a threshold of 4 updates.

The dynamic hybrid protocol reduces the degree of sharing in applications significantly. Taking the average number of updates sent per write to shared data as our metric, we see that the dynamic hybrid strategy provides reductions in the degree of sharing for 256-byte blocks of as much as factors of 4.8, 2.3, 2.4, 8.4, 6.8, and 5.0 for `Barnes-Hut`, `CG`, `Em3d`, `New Mp3d`, `SOR`, and `Blocked LU` respectively, compared to pure WU and the other strategies for improving that protocol. Reducing the degree of sharing is an important characteristic of this strategy, as the performance of directory schemes based on a limited number of pointers degrades quickly when the hardware pointers are frequently exhausted.

Comparing figures 16 and 17 with 27 and 28 we can see that the dynamic hybrid protocol is very effective at reducing the coherence traffic associated with the pure WU protocol. Depending on the application, either the dynamic or the static hybrid protocols generate the least amount of update traffic among all techniques we study. Comparing the traffic categorizations for `SC Blocked LU` and `Blocked LU` under the dynamic hybrid protocol, we find that the dynamic hybrid strategy entails significantly more (useless) updates. Coalescing generates less total traffic than the dynamic hybrid protocol for all applications; `CG` exhibits the largest difference: 12-31%, depending on the block size.

Software caching and the dynamic hybrid protocol are the most successful strategies for reducing the useless coherence traffic associated with the pure WU protocol, but, in some cases, the latter technique also reduces the number of useful updates. A reduction in the number of useful updates

[3]The Alpha implementation uses the presence of the block in the primary or secondary cache as an implicit counter.

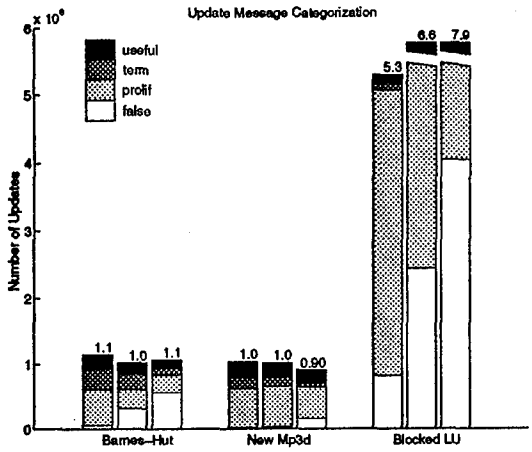

Figure 27: Categorization of Updates for Group 1 under Dynamic Hybrid Protocol.

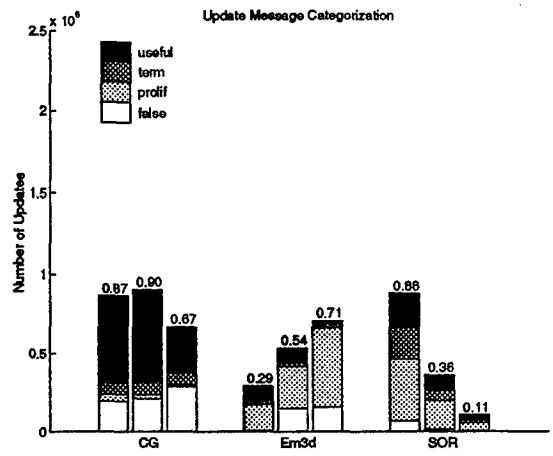

Figure 28: Categorization of Updates for Group 2 under Dynamic Hybrid Protocol.

indicates that the protocol is forcing processors to drop copies of blocks they will need later. Dropping such blocks may or may not cause performance degradation, depending on a tradeoff between the number and impact of useless updates that would have resulted in not invalidating the blocks and the higher miss rate.

Figure 29 compares the read miss rates of our programs under WI (left) and under the dynamic hybrid protocol (right). Our categorization includes a new class of miss (labeled "Drop" in the figure), which occurs when a processor takes a miss on a block that was previously in the cache, but was invalidated when its counter reached the competitive threshold. The figure shows that `CG`, `New Mp3d`, `SOR`, and `Blocked LU` suffer significantly from bad invalidation decisions, while the two other programs are either mildly affected or not at all. The block size affects the drop miss rate of our applications in different ways. For the applications with excellent locality (`CG` and `SOR`), the number of drop misses decreases with an increase in block size, while for the other applications it either increases (`Em3d` and `Blocked LU`) or remains roughly the same (`New Mp3d` and `Barnes-Hut`).

Even though drop misses increase the miss rate of `New Mp3d`, this application obtains significant running time improvements under the dynamic hybrid protocol in comparison to pure WU: with 256-byte blocks, for instance, 2%, 19%, and 48% improvements were found with medium, high, and infinite bandwidth, respectively. A comparison against WI can be found in figure 30. The figure shows that `New Mp3d` performs somewhat better under the dynamic hybrid protocol than under WI for all bandwidth levels and block sizes. Albeit the good results for `New Mp3d`, the dynamic hybrid strategy do not achieve performance improvements for the other applications in our suite.

The poor miss rate performance of four of our applications under the dynamic hybrid protocol can be credited to the fact that, if programs exhibit a high degree of spatial locality of writes, 4 updates without intervention from the local processor is too small of a threshold, especially for the larger block sizes. One extreme option would have been to set the threshold to the number of words in a block plus 1. However, in the absence of write locality, this threshold would have entailed a large number of useless updates in the case of the large cache blocks. We opted for the threshold of 4 in order to reduce the number of useless updates. A combination of coalescing and the dynamic

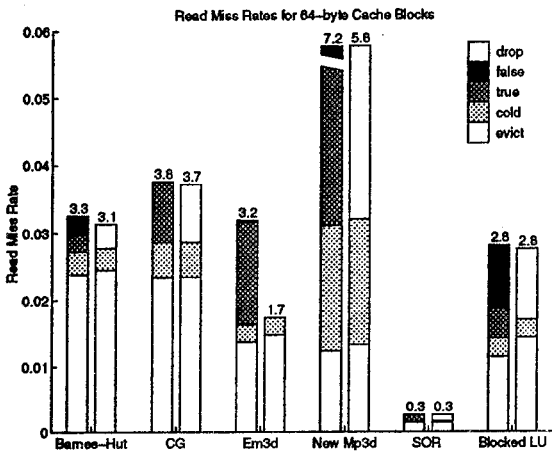

Figure 29: Miss Rate Under Dynamic Hybrid Protocol.

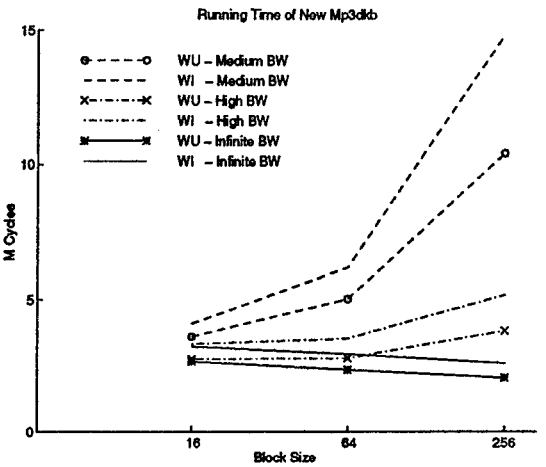

Figure 30: Running Time of `New Mp3d` Under Dynamic Hybrid Protocol.

hybrid protocol should diminish the importance of the competitive threshold. We study this and other combinations of strategies in the next section.

5.5 Combining Techniques

Our results so far show that coalescing write buffers provide significant reductions in coherence traffic accompanied by consistent performance improvements under the higher levels of bandwidth and moderately-sized cache blocks. The other techniques, although also successful at reducing the total network traffic, don't always deliver performance improvements. In this section, we investigate combinations of coalescing with the other techniques.

Under the combination of coalescing and software caching, WU matches the performance of WI for `SC Blocked LU`, as the program improves by 8-16% under WU, depending on the block size, and the WI performance remains roughly unaltered.

The combination of coalescing and the static hybrid protocol improved on the performance of the static hybrid technique in isolation in all cases. The combination of coalescing and the dynamic hybrid protocol (still with a competitive threshold of 4 updates) frequently improves on the performance of the dynamic hybrid strategy in isolation, while it occasionally improves on the performance of coalescing. Coalescing eliminates the problems associated with the specific value of the threshold, while the dynamic hybrid optimization reduces the degree of sharing in the program. For programs such as `New Mp3d` and `CG`, the reduced sharing markedly improves the performance of coalescing for the larger cache blocks under the practical levels of bandwidth.

In summary, all techniques we studied and their combinations were very successful at reducing the amount of useless coherence traffic generated by pure WU protocols. Software caching and the dynamic hybrid protocol stand out as being able to significantly increase the percentage of useful updates in applications. Coalescing write buffers provided the greatest reductions in the total number of bytes transferred as well as the greatest improvements in performance. In only a few instances did another technique or combination of techniques outperform coalescing. The major performance

problem for coalescing buffers occurs when cache blocks are very large and the bandwidth in the system is not extraordinarily high.

5.6 Potential for Further Improvements to WU protocols

The traffic categorizations presented in the previous section clearly show that useless updates dominate the coherence traffic of our applications, even when techniques intended to reduce this type of traffic are applied. In this section, we comment on the potential for further reductions in the percentage of useless updates.

Our analysis of the sources of updates in applications shows that a large number of proliferation updates stems from a round-robin assignment of pages to processors. The problem with this type of mapping is that it is often the case that the home node is not one of the processors sharing the blocks for which it receives updates. Thus, a simple strategy that can further reduce the number of proliferation updates is to assure that one of the processors sharing a block is the home node for that block. In fact, the processor that writes to the block the most should be made the home of the block. Page placement and migration techniques, such as presented in [Chandra *et al.*, 1994; Marchetti *et al.*, 1994], can be used to implement this strategy successfully in many cases. If applied to SOR, for instance, this strategy would have eliminated about half of the useless updates in the program.

Another way in which proliferation updates can be eliminated is by combining writes to the same words in software. This optimization can be implemented at the compiler level, by having the compiler use registers for shared data writes inside of critical sections, and only issuing writes to memory at the end of the section. Our experiments with SC Blocked LU implemented this strategy at the application level; throughout the program writes to global data were only issued when their final values had been computed. This scheme proved extremely useful for improving Blocked LU, but did not appear applicable to the other applications we studied.

Flushing widely shared cache blocks at the end of critical sections can also greatly reduce the number of proliferation updates. Centralized counters, locks, and barriers are examples of data structures that cause a large number of useless updates. Distributing those data structures should improve performance even more significantly however, as it would avoid the increased read latency generated by block flushes.

6 Protocol Performance on Future Multiprocessors

Our results so far have shown that WU performs at least as well as WI for *all* applications and combinations of bandwidth and block size we consider, provided that techniques for reducing the amount of useless update traffic are applied. Under infinite bandwidth, the performance advantage of the update-based protocols over WI ranges from a few percent for the applications dominated by replacement misses (Barnes-Hut and CG) and applications with extremely small miss rates (SC Blocked LU and SOR) up to 23% for the applications dominated by coherence misses (New Mp3d and Em3d).

Note however that our previous results are based on current architectural assumptions, which do not favor update-based protocols in many respects: replacement misses (as opposed to sharing-related misses) dominate the miss rate of many applications, network and memory latencies are

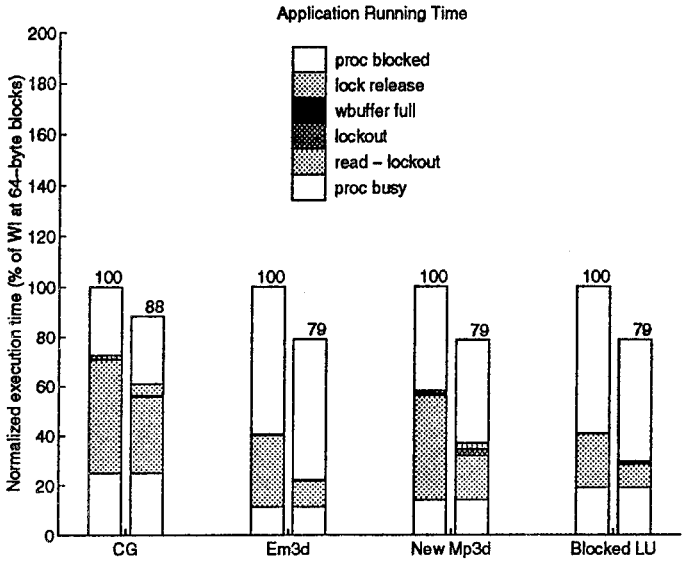

Figure 31: Running Time on Next-Generation Architecture.

relatively low considering the latest advances in superscalar microprocessors, the highest practical level of bandwidth is relatively low considering the amount of data an update-based protocol has to tackle, and our memory bus assumptions are such that a coalesced update with one dirty word takes as long to complete as if all the words in the (possibly large) cache block were dirty.

We now extrapolate the current architectural trends in order to explore the WI versus WU issue in the context of a more aggressive scalable multiprocessor design. In order to quantify the impact of future architectures on protocol performance, we increase the size of caches to 128K bytes, double the memory and network latency and bandwidth, and simulate a memory bus that can handle variable length update operations.

Figure 31 presents the WI (left) and WU (right) running times of 4 of our applications on our next-generation architecture with coalescing write buffers and 64-byte blocks. For all of our applications, including the ones not shown in the figure, the running time difference between the protocols increased under our more aggressive assumptions. Comparing our previous results for high bandwidth and coalescing against our new results, we see that the performance improvement of `CG` for 64-byte blocks goes from 8% to 12%, while the improvements of **`New Mp3d`**, **`Blocked LU`**, and **`Em3d`** go from 8% to 21%, 5% to 21%, and 17% to 21%, respectively. These results suggest that the architectural trends we have been observing should increase the performance advantage of WU over WI significantly in the future.

7 Conclusions

Our simulations of WU and WI coherence protocols for scalable multiprocessors showed that, although WU produces a lower miss rate, the enormous network traffic generated by WU degrades the running time of some of our applications. Even infinite bandwidth is not enough to enable WU to significantly outperform WI on all of our applications. We found the cause of the poor WU performance to be that the excessive number of update transactions in a pure WU protocol generates network and memory congestion, which is reflected in various forms of performance degradation.

To alleviate the network traffic generated by WU, we classified updates into useful and useless categories, and showed that a vast majority of the updates are useless; in most cases, useless updates are more than 90% of the total number of updates. By relating the useless updates back to the source code, we determined the application characteristics that cause useless updates and evaluated several software and hardware techniques for eliminating them. These techniques include coalescing write buffers, dynamic and static hybrid protocols, and software caching.

We studied the isolated and combined effect of these techniques on our categorization of the coherence traffic and on application execution time. Our results showed that software caching and the dynamic hybrid protocol are the most successful strategies for eliminating useless updates, and that software caching and coalescing write buffers produce the lowest amount of traffic by merging multiple updates in a single message. In terms of execution time, coalescing write buffers exhibit the most consistent improvements. Coalescing improves WU enormously, but only slightly improves WI. Wide coalescing write buffers were shown unnecessary for WU and may even cause serious performance degradation in the presence of relatively low bandwidth. The combination of coalescing and the dynamic hybrid protocol also performed well. Coalescing improves the dynamic hybrid strategy by reducing the importance of the specific value of the competitive threshold, while self-invalidating of cache blocks reduces the degree of sharing in application programs.

Although the techniques we considered significantly reduce the useless traffic associated with update-based protocols, a large number of useless updates remains. Based on that observation, we suggested several directions for further eliminating useless traffic in update-based protocols, such as flushing widely shared cache blocks at the end of critical sections to reduce the number of proliferation updates found in applications.

Finally, our experiments showed that WU (with optimizations) performs better than our WI implementation for all of our applications. Current architectural trends (faster processors, longer latency, higher bandwidth) significantly magnify the performance difference between the two types of protocols in favor of WU.

Acknowledgements

We would like to thank Leonidas Kontothanassis for numerous discussions on the topic of this work. We would also like to thank the Alewife group at MIT for the `CG` application and their detailed network simulator.

References

[Archibald and Baer, 1986] James Archibald and Jean-Loup Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[Bailey et al., 1994] D. Bailey et al., "The NAS Parallel Benchmarks," Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[Bianchini and Kontothanassis, 1994] R. Bianchini and L. I. Kontothanassis, "Algorithms for Categorizing Multiprocessor Communication Under Invalidate and Update-Based Coherence Protocols," Technical Report 533, Department of Computer Science, University of Rochester, September 1994.

[Bianchini and LeBlanc, 1992] R. Bianchini and T. J. LeBlanc, "Software Caching on Cache-Coherent Multiprocessors," In *Proceedings of the 4th Symposium on Parallel and Distributed Processing*, Dallas, TX, December 1992.

[Bisiani and Ravishankar, 1990] R. Bisiani and M. Ravishankar, "Plus: A Distributed Shared-Memory System," In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[Carter *et al.*, 1991] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," In *Proceedings of the 13th Symposium on Operating Systems Principles*, October 1991.

[Chandra *et al.*, 1994] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and Page Migration for Multiprocessor Compute Servers," In *Proceedings of the 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[Dackland *et al.*, 1992] K. Dackland, E. Elmroth, B. Kagstrom, and C. Van Loan, "Parallel Block Matrix Factorizations on the Shared-Memory Multiprocessor IBM 3090 VF/600J," *The International Journal of Supercomputer Applications*, 6(1):69–97, Spring 1992.

[Dahlgren *et al.*, 1994] F. Dahlgren, M. Dubois, and P. Stenstrom, "Combined Performance Gains of Simple Cache Protocol Extensions," In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[Dahlgren and Stenstrom, 1994] F. Dahlgren and P. Stenstrom, "Reducing the Write Traffic for a Hybrid Cache Protocol," In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

[Dubois *et al.*, 1993] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom, "The Detection and Elimination of Useless Misses in Multiprocessors," In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 88–97, San Diego, CA, May 1993.

[Eggers and Jeremiassen, 1991] S. J. Eggers and T. E. Jeremiassen, "Eliminating False Sharing," In *Proceedings 1991 International Conference on Parallel Processing*, pages 377–381, St. Charles, IL, August 1991.

[Eggers and Katz, 1988] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation," In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373–383, May 1988.

[Goodman, 1983] James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983.

[Jouppi, 1993] Norman P. Jouppi, "Cache Write Policies and Performance," In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 191–201, May 1993.

[Karlin *et al.*, 1988] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive Snoopy Caching," *Algorithmica*, 3:79–119, 1988.

[Lenoski *et al.*, 1990] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, May 1990.

[Lenoski *et al.*, 1992] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy, "The DASH Prototype: Implementation and Performance," In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[Marchetti *et al.*, 1994] M. Marchetti, L. I. Kontothanassis, R. Bianchini, and M. L. Scott, "Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems," Technical Report 535, Department of Computer Science, University of Rochester, September 1994.

[McCreight, 1984] E. M. McCreight, "The Dragon Computer System, an Early Overview," In *NATO Advanced Study Insitiute on Microarchitecture of VLSI Computers*, July 1984.

[Papamarcos and Patel, 1984] Mark S. Papamarcos and Janak H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[Singh *et al.*, 1992] J.P. Singh, W-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, 20(1):5–44, March 1992.

[Thacker *et al.*, 1992] Charles P. Thacker, David G. Conroy, and Lawrence C. Stewart, "The Alpha Demonstration Unit: A High-performance Multiprocessor for Software and Chip Development," *Digital Technical Journal*, 4(4):51–65, 1992.

[Thacker and Stewart, 1987] Charles P. Thacker and Lawrence C. Stewart, "Firefly: a Multiprocessor Workstation," In *Proceedings of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 164–172, Palo Alto, CA, October 1987.

[Torellas *et al.*, 1990] J. Torellas, M. S. Lam, and J. L. Henessy, "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates," In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[Veenstra and Fowler, 1992] J. E. Veenstra and R. J. Fowler, "A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols," In *Proceedings of the 5rd International Conference on Architectual Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[Veenstra and Fowler, 1994] J. E. Veenstra and R. J. Fowler, "The Prospects for On-Line Hybrid Coherency Protocols on Bus-Based Multiprocessors," Technical Report 490, Department of Computer Science, University of Rochester, March 1994.

[Wilson and LaRowe, 1992] A. W. Wilson and R. P. LaRowe, "Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture," *Journal of Parallel and Distributed Computing*, 15(4):351–367, 1992.